Introduction to the Standard Template Library

14

Key Concepts

Software evolution | Standard templates | Standard C++ library | Containers | Sequence containers | Associative containers | Derived containers | Algorithms | Iterators | Function object

14.1 Introduction

We have seen how templates can be used to create generic classes and functions that could extend support for generic programming. In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett-Packard developed a set of general-purpose templatized classes (data structures) and functions (algorithms) that could be used as a standard approach for storing and processing of data. The collection of these generic classes and functions is called the *Standard Template Library (STL).* The STL has now become a part of the ANSI standard C++ class library.

STL is large and complex and it is difficult to discuss all of its features in this chapter. We therefore present here only the most important features that would enable the readers to begin using the STL effectively. Using STL can save considerable time and effort, and lead to high quality programs. All these benefits are possible because we are basically “reusing” the well-written and well-tested components defined in the STL.

STL components which are now part of the Standard C++ Library are defined in the **namespace std.** We must therefore use the **using namespace** directive

```
using namespace std;
```

to inform the compiler that we intend to use the Standard C++ Library. All programs in this chapter use this directive.

14.2 Components of STL

The STL contains several components. But at its core are three key components. They are:

- containers,
- algorithms, and
- iterators.

These three components work in conjunction with one another to provide support to a variety of programming solutions. The relationship between the three components is shown in Fig. 14.1. *Algorithms* employ *iterators* to perform operations stored in *containers.*

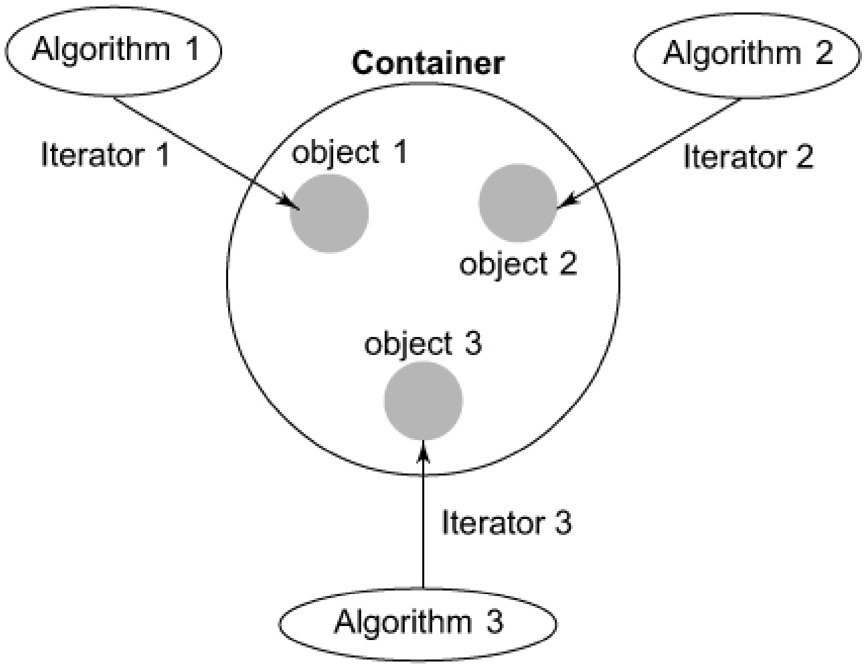

Fig. 14.1 *Relationship between the three STL components*

A *container* is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

An *algorithm* is a procedure that is used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, copying, sorting, and merging. Algorithms are implemented by template functions.

An *iterator* is an object (like a pointer) that points to an element in a container. We can use iterators to move through the contents of containers. Iterators are handled just like pointers. We can increment or decrement them. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.

14.3 Containers

As stated earlier, containers are objects that hold data (of same type). The STL defines ten containers which are grouped into three categories as illustrated in Fig. 14.2. Table 14.1 gives the details of all these containers as well as header to be included to use each one of them and the type of iterator supported by each container class.

Table 14.1 *Containers supported by the STL*

Container	*Description*	*Header file*	*Iterator*
vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element	<vector>	Random access
list	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional

deque	A double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access
set	An associate container for storing unique sets. Allows rapid lookup. (No duplicates allowed)	<set>	Bidirectional
multiset	An associate container for storing non-unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing unique key/value pairs. Each key is associated with only one value (One-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional
stack	A standard stack. Last-in-first-out(LIFO)	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO).	<queue>	No iterator
priority– queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

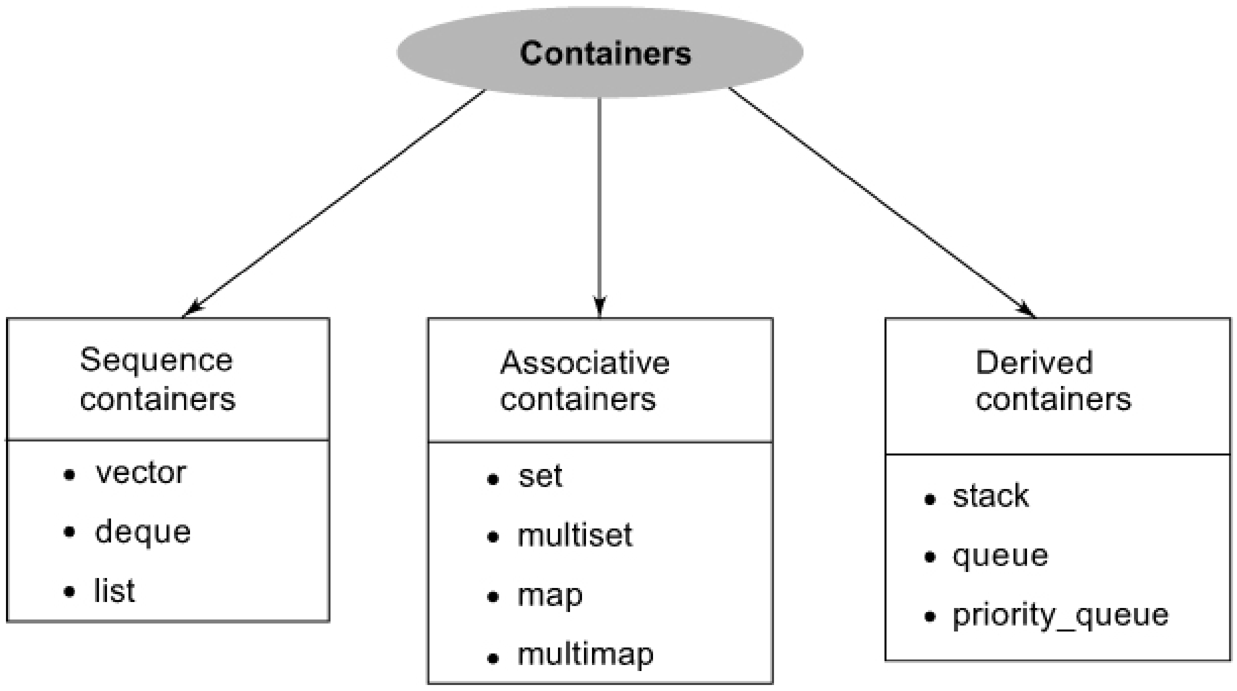

Fig. 14.2 *Three major categories of containers*

Each container class defines a set of functions that can be used to manipulate its contents. For example, a vector container defines functions for inserting elements, erasing the contents, and swapping the contents of two vectors.

Sequence Containers

Sequence containers store elements in a linear sequence, like a line as shown in Fig. 14.3. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operations on them.

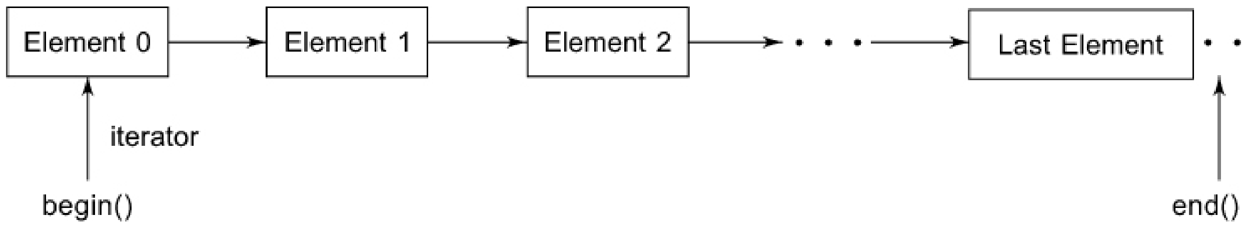

Fig. 14.3 *Elements in a sequence container*

The STL provides three types of sequence containers:

- vector
- list
- deque

Elements in all these containers can be accessed using an iterator. The difference between the three of them is related to only their performance. Table 14.2 compares their performance in terms of speed of random access and insertion or deletion of elements.

Table 14.2 *Comparison of sequence containers*

Container	*Random access*	*Insertion or deletion in the middle*	*Insertion or deletion at the ends*
vector	Fast	Slow	Fast at back
list	Slow	Fast	Fast at front
deque	Fast	Slow	Fast at both the ends

Associative Containers

Associative containers are designed to support direct access to elements using keys. They are not sequential. There are four types of associative containers:

- set
- multiset
- map
- multimap

All these containers store data in a structure called *tree* which facilitates fast searching, deletion, and insertion. However, these are very slow for random access and inefficient for sorting.

Containers **set** and **multiset** can store a number of items and provide operations for manipulating them using the values as the *keys.* For example, a **set** might store objects of the **student** class which are ordered alphabetically using names as keys. We can search for a desired student using his name as the key. The main difference between a **set** and a **multiset** is that a **multiset** allows duplicate items while a **set** does not.

Containers **map** and **multimap** are used to store pairs of items, one called the *key* and the other called the *value.* We can manipulate the values using the keys associated with them. The values are sometimes called *mapped values.* The main difference between a **map** and a **multimap** is that a **map** allows only one key for a given value to be stored while **multimap** permits multiple keys.

Derived Containers

The STL provides three derived containers namely, **stack, queue,** and **priority_queue.** These are also known as *container adaptors.*

Stacks, queues and priority queues can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions **pop()** and **push()** for implementing deleting and inserting operations.

14.4 Algorithms

Algorithms are functions that can be used generally across a variety of containers for processing their contents. Although each container provides functions for its basic operations, STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithms also permit us to work with two different types of containers at the same time. Remember, STL algorithms are not member functions or friends of containers. They are standalone template functions.

STL algorithms reinforce the philosophy of reusability. By using these algorithms, programmers can save a lot of time and effort. To have access to the STL algorithms, we must include **<algorithm>** in our program.

STL algorithms, based on the nature of operations they perform, may be categorized as under:

- Retrieve or nonmutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

These algorithms are summarized in Tables 14.3 to 14.7. STL also contains a few numeric algorithms under the header file **<numeric>.** They are listed in Table 14.8.

Table 14.3 *Nonmutating algorithms*

Operations	*Description*
adjacent_find( )	Finds adjacent pair of objects that are equal
count( )	Counts occurrence of a value in a sequence
count_if( )	Counts number of elements that matches a predicate
equal( )	True if two ranges are the same
find( )	Finds first occurrence of a value in a sequence
find_end( )	Finds last occurrence of a value in a sequence
find_first_of( )	Finds a value from one sequence in another
find_if( )	Finds first match of a predicate in a sequence
for_each( )	Apply an operation to each element
mismatch( )	Finds first elements for which two sequences differ

search( )	Finds a subsequence within a sequence
search_n( )	Finds a sequence of a specified number of similar elements

Table 14.4 *Mutating algorithms*

Operations	*Description*
Copy( )	Copies a sequence
copy_backward( )	Copies a sequence from the end
fill( )	Fills a sequence with a specified value
fill_n( )	Fills first n elements with a specified value
generate( )	Replaces all elements with the result of an operation
generate_n( )	Replaces first n elements with the result of an operation
iter_swap( )	Swaps elements pointed to by iterators
random_shuffle( )	Places elements in random order
remove( )	Deletes elements of a specified value
remove_copy( )	Copies a sequence after removing a specified value
remove_copy_if( )	Copies a sequence after removing elements match- ing a predicate
remove_if( )	Deletes elements matching a predicate
replace( )	Replaces elements with a specified value
replace_copy( )	Copies a sequence replacing elements with a given value
replace_copy_if( )	Copies a sequence replacing elements matching a predicate
replace_if( )	Replaces elements matching a predicate

reverse( )	Reverses the order of elements
reverse_copy( )	Copies a sequence into reverse order
rotate( )	Rotates elements
rotate_copy( )	Copies a sequence into a rotated
swap( )	Swaps two elements
swap_ranges( )	Swaps two sequences
transform( )	Applies an operation to all elements
unique( )	Deletes equal adjacent elements
unique_copy( )	Copies after removing equal adjacent elements

Table 14.5 *Sorting algorithms*

Operations	*Description*
binary_search( )	Conducts a binary search on an ordered sequence
equal_range( )	Finds a subrange of elements with a given value
inplace_merge( )	Merges two consecutive sorted sequences
lower_bound( )	Finds the first occurrence of a specified value
make_heap( )	Makes a heap from a sequence
merge( )	Merges two sorted sequences
nth_element( )	Puts a specified element in its proper place
partial_sort( )	Sorts a part of a sequence
partial_sort_copy()	Sorts a part of a sequence and then copies
Partition( )	Places elements matching a predicate first
pop_heap( )	Deletes the top element
push_heap( )	Adds an element to heap
sort( )	Sorts a sequence
sort_heap( )	Sorts a heap

stable_partition( )	Places elements matching a predicate first matching relative order
stable_sort( )	Sorts maintaining order of equal elements
upper_bound( )	Finds the last occurrence of a specified value

Table 14.6 *Set algorithms*

Operations	*Description*
includes( )	Finds whether a sequence is a subsequence of another
set_difference( )	Constructs a sequence that is the difference of two ordered sets
set_intersection( )	Constructs a sequence that contains the intersection of ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union( )	Produces sorted union of two ordered sets

Table 14.7 *Relational algorithms*

Operations	*Description*
equal( )	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max( )	Gives maximum of two values
max_element( )	Finds the maximum element within a sequence
min( )	Gives minimum of two values
min_element( )	Finds the minimum element within a sequence

mismatch( )	Finds the first mismatch between the elements in two sequences

Table 14.8 *Numeric algorithms*

Operations	*Description*
accumulate( )	Accumulates the results of operation on a sequence
adjacent_difference( )	Produces a sequence from another sequence
inner_product( )	Accumulates the results of operation on a pair of sequences
partial_sum( )	Produces a sequence by operation on a pair of sequences

14.5 Iterators

Iterators behave like pointers and are used to access container elements. They are often used to traverse from one element to another, a process known as *iterating* through the container.

There are five types of iterators as described in Table 14.9.

Table 14.9 *Iterators and their characteristics*

Iterator	*Access method*	*Direction of movement*	*I/O capability*	*Remark*
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

Different types of iterators must be used with the different types of containers (See Table 14.1). Note that only sequence and associative containers are traversable with iterators.

Each type of iterator is used for performing certain functions. Figure 14.4 gives the functionality Venn diagram of the iterators. It illustrates the level of functionality provided by different categories of iterators.

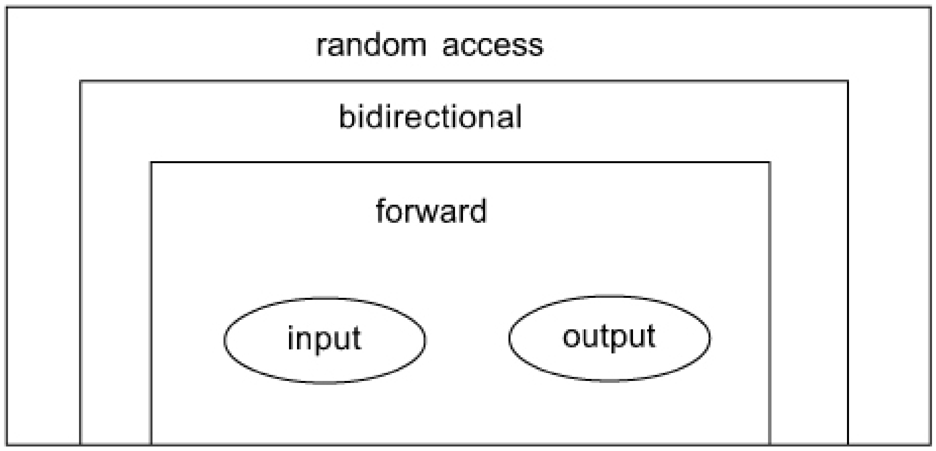

Fig. 14.4 *Functionality Venn diagram of iterators*

The *input* and *output* iterators support the least functions. They can be used only to traverse in a container. The *forward* iterator supports all operations of input and output iterators and also retains its position in the container. A *bidirectional* iterator, while supporting all forward iterator operations, provides the ability to move in the backward direction in the container. A *random* access iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location. Table 14.10 summarizes the operations that can be performed on each iterator type.

Table 14.10 *Operations supported by iterators*

Iterator	*Element access*	*Read*	*Write*	*Increment operation*	*Comparison*
Input	->	v = *p		++	==, !=
Output			*p = v	++	
Forward	->	v = *p	*p = v	++	==, !=
Bidirectional	->	v = *p	*p = v	++, −−	==, !=
Random access	->, [ ]	v = *p	*p = v	++, −−, +, −, +=, −=	==, !=, <, >, <=, >=

14.6 Application of Container Classes

It is beyond the scope of this book to examine all the containers supported in the STL and provide illustrations. Therefore, we illustrate here the use of the three most popular containers, namely, **vector, list,** and **map.**

Vectors

The **vector** is the most widely used container. It stores elements in contiguous memory locations and enables direct access to any element using the subscript operator [ ]. A **vector** can change its size dynamically and therefore allocates memory as needed at run time.

The **vector** container supports random access iterators, and a wide range of iterator operations (See Table 14.10) may be applied to a **vector** iterator. Class **vector** supports a number of constructors for creating **vector** objects.

```
    vector<int> v1;                     // Zero-length int vector
    vector<double> v2(10);                  // 10-element double
vector
    vector<int> v3(v4);                  // Creates v3 from v4
    vector<int> v(5, 2);                // 5-element vector of 2s
```

The **vector** class supports several member functions as listed in Table 14.11. We can also use all the STL algorithms on a **vector.**

Table 14.11 *Important member functions of the vector class*

Function	*Task*
at( )	Gives a reference to an element
back( )	Gives a reference to the last element
begin( )	Gives a reference to the first element
capacity( )	Gives the current capacity of the vector
clear( )	Deletes all the elements from the vector
empty( )	Determines if the vector is empty or not
end( )	Gives a reference to the end of the vector
erase( )	Deletes specified elements
insert( )	Inserts elements in the vector
pop_back( )	Deletes the last element
push_back( )	Adds an element to the end
resize( )	Modifies the size of the vector to the specified value
size( )	Gives the number of elements
swap( )	Exchanges elements in the specified two vectors

Program 14.1 illustrates the use of several functions of the **vector** class template. Note that an iterator is used as a pointer to elements of the vector. We must include header file **<vector>** to use **vector** class in our programs.

Program 14.1 Using Vectors

```
#include <iostream>
#include <vector>                          // Vector header file

using namespace std;
```

```
void display(vector<int> &v)
{
        for(int i=0;i<v.size();i++)
        {
             cout << v[i] << " ";
        }
        cout << "\n";
}

int main()
{
vector<int> v;                  // Create a vector of type int
cout << "Initial size = " << v.size() << "\n";
// Putting values into the vector
int x;
cout << "Enter five integer values: ";
for(int i=0; i<5; i++)
{
          cin >> x;
          v.push_back(x);
}
cout << "Size after adding 5 values: ";
cout << v.size() << "\n";

// Display the contents
cout << "Current contents: \n";
display(v);

// Add one more value
v.push_back(6.6);                       // float value truncated to int

// Display size and contents
cout << "\nSize = " << v.size() << "\n";
cout << "Contents now: \n";
display(v);

// Inserting elements
vector<int> :: iterator itr = v.begin();   // iterator
itr = itr + 3;                              // itr points to 4th element
v.insert(itr,1,9);

// Display the contents
cout << "\nContents after inserting: \n";
display(v);

// Removing 4th and 5th elements
v.erase(v.begin()+3,v.begin()+5);       // Removes 4th and 5th
element

// Display the contents
cout << "\nContents after deletion: \n";
display(v);
cout << "END\n";
return(0);
}
```

The output of Program 14.1 would be:

```
Initial size = 0

Enter five integer values: 1 2 3 4 5
Size after adding 5 values: 5
Current contents:
1 2 3 4 5

Size = 6
Contents now:
1 2 3 4 5 6

Contents after inserting:
1 2 3 9 4 5 6

Contents after deletion:
```

```
1 2 3 5 6
END
```

The program uses a number of functions to create and manipulate a vector. The member function **size()** gives the current size of the vector. After creating an **int** type empty vector **v** of zero size, the program puts five values into the vector using the member function **push_back().** Note that **push_ back()** takes a value as its argument and adds it to the back end of the vector. Since the vector **v** is of type **int,** it can accept only integer values and therefore the statement

```
v.push_back(6.6);
```

truncates the values 6.6 to 6 and then puts it into the vector at its back end.

The program uses an iterator to access the vector elements. The statement

```
vector<int> :: iterator itr = v.begin();
```

declares an iterator **itr** and makes it to point to the first position of the vector. The statements

```
itr = itr + 3;
v.insert(itr,9);
```

inserts the value 9 as the fourth element. Similarly, the statement

```
v.erase(v.begin()+3, v.begin()+5);
```

deletes 4th and 5 th elements from the vector . Note that **erase(m,n)** deletes only n-m elements starting from mth element and the nth element is not deleted.

The elements of a vector may also be accessed using subscripts (as we do in arrays). Notice the use of **v[i]** in the function **display()** for displaying the contents of **v.** The call **v.size()** in the **for** loop of **display()** gives the current size of **v.**

Lists

The **list** is another container that is popularly used. It supports a bidirectional, linear list and provides an efficient implementation for deletion and insertion operations. Unlike a vector, which supports random access, a list can be accessed sequentially only.

Bidirectional iterators are used for accessing list elements. Any algorithm that requires input, output, forward, or bidirectional iterators can operate on a **list.** Class **list** provides many member functions for manipulating the elements of a list. Important member functions of the **list** class are given in Table 14.12. Use of some of these functions is illustrated in Program 14.2. Header file **<list>** must be included to use the container class **list.**

Program 14.2 Using Lists

```
#include <iostream>
#include <list>
#include <cstdlib>                    // For using rand() function

using namespace std;

void display(list<int> &lst)
{
    list<int> :: iterator p;
    for(p = lst.begin(); p != lst.end(); ++p)
         cout << *p << ", ";
    cout << "\n\n";
}

int main()
{
    list<int> list1;              // Empty list of zero length
    list<int> list2(5);            // Empty list of size 5
```

```
 for(int i=0;i<3;i++)
         list1.push_back(rand()/100);

 list<int> :: iterator p;
 for(p=list2.begin(); p!=list2.end();++p)
         *p = rand()/100;
 cout << "List1 \n";
 display(list1);
 cout << "List2 \n";
 display(list2);

// Add two elements at the ends of list1
 list1.push_front(100);
 list1.push_back(200);

// Remove an element at the front of list2
 list2.pop_front();

 cout << "Now List1 \n";
 display(list1);
 cout << "Now List2 \n";
 display(list2);

 list<int> listA, listB;
 listA = list1;
 listB = list2;

// Merging two lists(unsorted)
 list1.merge(list2);
 cout << "Merged unsorted lists \n";
 display(list1);

// Sorting and merging
 listA.sort();
 listB.sort();
 listA.merge(listB);
```

```
    cout << "Merged sorted lists \n";
    display(listA);

   // Reversing a list
    listA.reverse();
    cout << "Reversed merged list \n";
    display(listA);

    return(0);
}
```

The output of Program 14.2 would be:

```
List1
0, 184, 63,

List2
265, 191, 157, 114, 293,

Now List1
100, 0, 184, 63, 200,

Now List2
191, 157, 114, 293,

Merged unsorted lists
100, 0, 184, 63, 191, 157, 114, 200, 293,

Merged sorted lists
0, 63, 100, 114, 157, 184, 191, 200, 293,

Reversed merged list
293, 200, 191, 184, 157, 114, 100, 63, 0,
```

The program declares two empty lists, **list1** with zero length and **list2** of size 5. The **list1** is filled with three values using the member function **push_back()** and math function rand(). The **list2** is filled using a **list** type iterator **p** and a **for** loop. Remember that **list2.begin()** gives the position of the first element while **list2.end()** gives the position immediately after the last element. Values are inserted at both the ends using **push_front()** and **push_back()** functions. The function **pop_front()** removes the first element in the list. Similarly, we may use **pop_back()** to remove the last element.

The objects of list can be initialized with other list objects like,

```
listA = list1;
listB = list2;
```

The statement

```
list1.merge(list2);
```

simply adds the **list2** elements to the end of **list1**. The elements in a list may be sorted in increasing order using **sort()** member function. Note that when two sorted lists are merged, the elements are inserted in appropriate locations and therefore the merged list is also a sorted one.

We use a **display()** function to display the contents of various lists. Note the difference between the implementations of **display()** in Program 14.1 and Program 14.2.

Table 14.12 *Important member functions of the list class*

Function	*Task*
back( )	Gives reference to the last element
begin( )	Gives reference to the first element
clear( )	Deletes all the elements
empty( )	Decides if the list is empty or not
end( )	Gives reference to the end of the list
erase( )	Deletes elements as specified
insert( )	Inserts elements as specified

merge( )	Merges two ordered lists
pop_back()	Deletes the last element
pop_front()	Deletes the first element
push_back()	Adds an element to the end
push_front( )	Adds an element to the front
remove( )	Removes elements as specified
resize( )	Modifies the size of the list
reverse( )	Reverses the list
size( )	Gives the size of the list
sort( )	Sorts the list
splice( )	Inserts a list into the invoking list
swap( )	Exchanges the elements of a list with those in the invoking list
unique( )	Deletes the duplicating elements in the list

Maps

A **map** is a sequence of (key, value) pairs where a single value is associated with each unique key as shown in Fig. 14.5. Retrieval of values is based on the key and is very fast. We should specify the key to obtain the associated value.

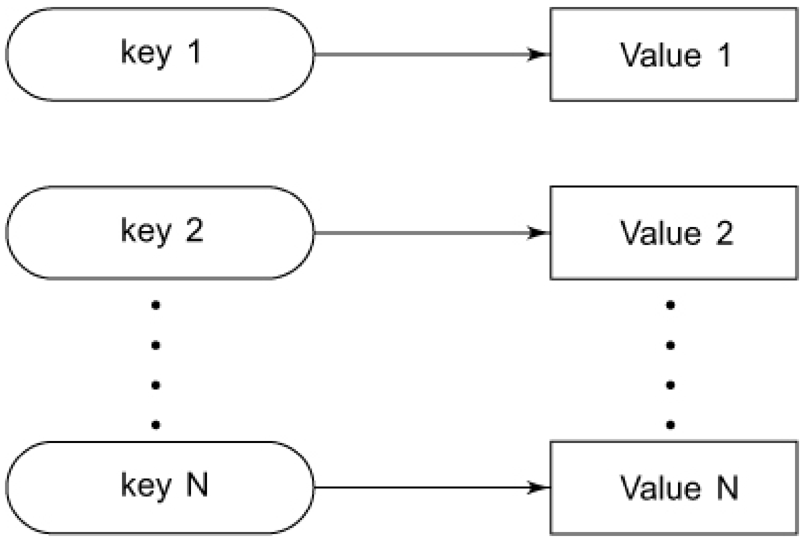

Fig. 14.5 *The key-value pairs in a map*

A **map** is commonly called an *associative array.* The key is specified using the subscript operator [ ] as shown below:

```
phone[ "John" ] = 1111;
```

This creates an entry for “John” and associates (i.e., assigns) the value 1111 to it. **phone** is a **map** object. We can change the value, if necessary, as follows:

```
phone[ "John" ] = 9999;
```

This changes the value 1111 to 9999. We can also insert and delete pairs anywhere in the **map** using **insert( )** and **erase( )** functions. Important member functions of the **map** class are listed in Table 14.13.

Table 14.13 *Important member functions of the map class*

Function	*Task*
begin( )	Gives reference to the first element
clear( )	Deletes all elements from the map
empty( )	Decides whether the map is empty or not

end( )	Gives a reference to the end of the map
erase( )	Deletes the specified elements
find( )	Gives the location of the specified element
insert( )	Inserts elements as specified
size( )	Gives the size of the map
swap( )	Exchanges the elements of the given map with those of the invoking map

Program 14.3 shows a simple example of a **map** used as an associative array. Note that **<map>** header must be included.

Program 14.3 Using Maps

```
#include <iostream>
#include <map>
#include <string>

using namespace std;
typedef map<string,int> phoneMap;

int main()
{
        string name;
        int number;
        phoneMap phone;
        cout << "Enter three sets of name and number \n";

        for(int i=0;i<3;i++)
        {
                cin >> name;        // Get key
                cin >> number;        // Get value
                phone[name] = number;        // Put them in map
        }

        phone["Jacob"] = 4444; // Insert Jacob

        phone.insert(pair<string,int> ("Bose", 5555));
        int n = phone.size();
        cout << "\nSize of Map: " << n << "\n\n";
        cout << "List of telephone numbers \n";
        phoneMap::iterator p;
        for(p=phone.begin(); p!=phone.end(); p++)
        {
        cout << (*p).first << " " << (*p).second << "\n";
        }
        cout << "\n";
        cout << "Enter name: ";          // Get name
        cin >> name;
        number = phone[name];            // Find number
        cout << "Number: " << number << "\n";
        return 0;
}
```

The output of Program 14.3 would be:

```
Enter three sets of name and number:
Prasanna 1111
Singh 2222
Raja 3333

Size of Map: 5

List of telephone numbers
Bose 5555
Jacob 4444
Prasanna 1111
Raja 3333
Singh 2222
```

```
Enter name: Raja
Number: 3333
```

The program first creates **phone** map interactively with three names and then inserts two more names into the map. Then, it displays all the names and their telephone numbers available in the map. Now the program requests the user to enter the name of a person. The program looks into the map, using the person name as a key, for the associated number and then prints the number.

NOTE: *That the names are printed in alphabetical order, although the original data was not. The list is automatically sorted using the key. In our example, the key is the name of person.*

We can access the two parts of an entry using the members **first** and **second** with an iterator of the **map** as illustrated in the program. That is,

```
(*p).first
```

gives the key, and

```
(*p).second
```

gives the value.

14.7 Function Objects

A function object is a function that has been wrapped in a class so that it looks like an object. The class has only one member function, the overloaded ( ) operator and no data. The class is templa-tized so that it can be used with different data types.

Function objects are often used as arguments to certain containers and algorithms. For example, the statement

```
sort(array, array+5, greater<int>());
```

uses the function object **greater<int>( )** to sort the elements contained in **array** in descending order.

Besides comparisons, STL provides many other predefined function objects for performing arithmetical and logical operations as shown in Table 14.14. Note that there are function objects corresponding to all the major C++ operators. For using function objects, we must include **<functional>** header file.

Table 14.14 *STL function objects in <functional>*

Function object	*Type*	*Description*
divides<T>	arithmetic	x/y
equal_to<T>	relational	x == y
greater<T>	relational	x > y
greater_equal<T>	relational	x >= y
less<T>	relational	x < y
less_equal<T>	relational	x <= y
logical_and<T>	logical	x && y
logical_not<T>	logical	!x
logical_or<T>	logical	x || y
minus<T>	arithmetic	x − y
modulus<T>	arithmetic	x % y
negate<T>	arithmetic	− x
not_equal_to<T>	relational	x != y
plus<T>	arithmetic	x + y
multiplies<T>	arithmetic	x * y

Note: The variables x and y represent objects of class T passed to the function object as arguments.

Program 14.4 illustrates the use of the function object **greater<>( )** in **sort( )** algorithm.

Program 14.4 Use of Function Objects in Algorithms

```
#include <iostream>
#include <algorithm>

#include <functional>
using namespace std;
int main()
{
        int x[] = {10,50,30,40,20};
        int y[] = {70,90,60,80};
        sort(x,x+5,greater<int>());
        sort(y,y+4);
        for(int i=0; i<5; i++)
        cout << x[i] << " ";
        cout << "\n";
        for(int j=0; j<4;
                cout << y[j] << " ";
        cout << "\n";
        int z[9];
        merge(x,x+5,y,y+4,z);
        for(i=0; i<9; i++)
                cout << z[i] << " ";
        cout << "\n";
        return(0);
}
```

The output of Program 14.4 would be:

```
50 40 30 20 10
60 70 80 90
50 40 30 20 10 60 70 80 90
```

NOTE: *The program creates two arrays **x** and **y** and initializes them with specified values. The program then sorts both of them using the algorithm **sort( )**. Note that **x** is sorted using the* *function object **greater<int>(**) and **y** is sorted without it and therefore the elements in **x** are in descending order.*

The program finally merges both the arrays and displays the content of the merged array. Note the form of **merge()** function and the results it produces.

Summary

- ❑ A collection of generic classes and functions is called the Standard Template Library (STL). STL components are part of C++ standard library.
- ❑ The STL consists of three main components: containers, algorithms, and iterators.
- ❑ Containers are objects that hold data of same type. Containers are divided into three major categories: sequential, associative, and derived.
- ❑ Container classes define a large number of functions that can be used to manipulate their contents.
- ❑ Algorithms are standalone functions that are used to carry out operations on the contents of containers, such as sorting, searching, copying, and merging.
- ❑ Iterators are like pointers. They are used to access the elements of containers thus providing a link between algorithms and containers. Iterators are defined for specific containers and used as arguments to algorithms.
- ❑ Certain algorithms use what are known as function objects for some operations. A function object is created by a class that contains only one overloaded operator () function.

Key Terms

<algorithm> | <cstdlib> | <deque> | <functional> | <list> | <map> | <numeric> | <queue> | <set> | <string> | <stack> | <vector> | algorithms **|** associative containers **|** bidirectional iterator **|** container adaptors **|** containers **| deque |** derived containers **|** forward iterator **|** function object **|** generic programming **|** input interator **|** iterators **|** keys **|** linear sequence **|** list **| map |** mapped values **| multimap |** multiple keys **| multiset |** mutating algorithms **|** namespace **|** nonmutating algorithms **|** numeric algorithms **|** output iterator **|** priority_queue **| queue |** random access iterator **|** relational algorithms **|** sequence containers **| set |** set algorithms **|** sorting algorithms **| stack |** standard C++ library **|** standard template library **|** templates **|** templatized classes **|** tree **|** using namespace **|** values **| vector**

Review Questions

14.1 What is STL? How is it different from the C++ Standard Library? Why is it gaining importance among the programmers?

14.2 List the three types of containers.

14.3 What is the major difference between a sequence container and an associative container?

14.4 What are the best situations for the use of the sequence containers?

14.5 What are the best situations for the use of the associative containers?

14.6 What is an iterator? What are its characteristics?

14.7 What is an algorithm? How STL algorithms are different from the conventional algorithms?

14.8 How are the STL algorithms implemented?

14.9 Distinguish between the following:

(a) lists and vectors

(b) sets and maps

(c) maps and multimaps

(d) queue and deque

(e) arrays and vectors

14.10 Compare the performance characteristics of the three sequence containers.

14.11 Suggest appropriate containers for the following applications:

(a) Insertion at the back of a container.

(b) Frequent insertions and deletion at both the ends of a container.

(c) Frequent insertions and deletions in the middle of a container.

(d) Frequent random access of elements.

14.12 State whether the following statements are true or false.

(a) An iterator is a generalized form of pointer.

(b) One purpose of an iterator is to connect algorithms to containers.

(c) STL algorithms are member functions of containers.

(d) The size of a vector does not change when its elements are removed.

(e) STL algorithms can be used with c-like arrays.

(f) An iterator can always move forward or backward through a container.

(g) The member function **end()** returns a reference to the last element in the container.

(h) The member function **back()** removes the element at the back of the container.

(i) The **sort()** algorithm requires a random-access iterator.

(j) A map can have two or more elements with the same key value.

Debugging Exercises

14.1 Identify the error in the following program.

```
#include <iostream.h>
#include <vector>

#define NAMESIZE 40

using namespace std;

class EmployeeMaster
{
    private:
        char name[NAMESIZE];
        int id;

    public:
        EmployeeMaster()
        {
            strcpy(name, "");
            id = 0;
        }
        EmployeeMaster(char name[NAMESIZE], int id)
            :id(id)
        {
            strcpy(this->name, name);
        }

        EmployeeMaster* getValuesFromUser()
        {
            EmployeeMaster *temp = new EmployeeMaster();
            cout << endl << "Enter user name : ";
            cin >> temp->name;
            cout << endl << "Enter user ID : ";
            cin << temp->id;
            return temp;
        }
        void displayRecord()
        {
            cout << endl << "Name : " << name;
            cout << endl << "ID : " << id << endl;
        }
};

void main()
{
        vector <EmployeeMaster*> emp;
        EmployeeMaster *temp = new EmployeeMaster();
        emp.push_back(getValuesFromUser());
        emp[0]->displayRecord();
        delete temp;

        temp = new EmployeeMaster("AlanKay", 3);
        emp.push_back(temp);
        emp[emp.capacity()]->displayRecord();
        emp[emp.size()]->displayRecord();
}
```

14.2 Identify the error in the following program.

```
#include <iostream>
#include <vector>
```

```
using namespace std;

int main()
{
    vector <int> v1;
    v1.push_back(10);
    v1.push_back(30);

    vector <int> v2;
    v2.push_back(20);
    v2.push_back(40);

    if(v1==v2)
        cout<<"vectors are equal";

    else

        cout<<"vectors are unequal\t";
        v1.swap(20);
    for(int y=0; y<v1.size(); y++)
    {
        cout<<"V1="<<v1[y]<<" ";
        cout<<"V2="<<v2[y]<<" ";
    }
    return 0;
}
```

14.3 Identify the error in the following program.

```
#include<iostream>
#include<list>

void main()
{

    list <int> l1;

    l1.push_front(10);
    l1.push_back(20);
    l1.push_front(30);
    l1.push_front(40);
    l1.push_back(10);
    l1.pop_front(40);

    l1.reverse ();
    l1.unique();
}
```

Programming Exercises

14.1 Write a code segment that does the following:

(a) Defines a vector **v** with a maximum size of 10.

(b) Sets the first element of **v** to 0.

(c) Sets the last element of **v** to 9.

(d) Sets the other elements to 1.

(e) Displays the contents of **v**. WEB

14.2 Write a program using the **find()** algorithm to locate the position of a specified value in a sequence container. WEB

14.3 Write a program using the algorithm **count()** to count how many elements in a container have a specified value. WEB

14.4 Create an array with even numbers and a list with odd numbers. Merge two sequences of numbers into a vector using the algorithm **merge()**. Display the vector.

14.5 Create a **student** class that includes a student's first name and his roll_number. Create five objects of this class and store them in a list thus creating a phone_lit. Write a program using this list to display the student name if the roll_number is given and vice-versa. WEB

14.6 Redo the Exercise 14.17 using a set.

14.7 A table gives a list of car models and the number of units sold in each type in a specified period. Write a program to store this table in a suitable container, and to display interactively the total value of a particular model sold, given the unit-cost of that model.

14.8 Write a program that accepts a shopping list of five items from the keyboard and stores them in a vector. Extend the program to accomplish the following:

(a) To delete a specified item in the list.

(b) To add an item at a specified location.

(c) To add an item at the end.

(d) To print the contents of the vector.